




OpenCV

4.8.0-dev



Open Source Computer Vision

- [Main Page](#)
- [Related Pages](#)
- [Modules](#)
- [+Namespaces](#)
- [+Classes](#)
- [+Files](#)
- [Examples](#)
- [Java documentation](#)
- 
- [Tutorials for contrib modules](#)
- [ArUco marker detection \(aruco module\)](#)

Detection of ArUco Markers

Next Tutorial: [Detection of ArUco Boards](#)

Pose estimation is of great importance in many computer vision applications: robot navigation, augmented reality, and many more. This process is based on finding correspondences between points in the real environment and their 2d image projection. This is usually a difficult step, and thus it is common to use synthetic or fiducial markers to make it easier.

One of the most popular approaches is the use of binary square fiducial markers. The main benefit of these markers is that a single marker provides enough correspondences (its four corners) to obtain the camera pose. Also, the inner binary codification makes them specially robust, allowing the possibility of applying error detection and correction techniques.

The aruco module is based on the [ArUco library](#), a popular library for detection of square fiducial markers developed by Rafael Muñoz and Sergio Garrido [\[89\]](#).

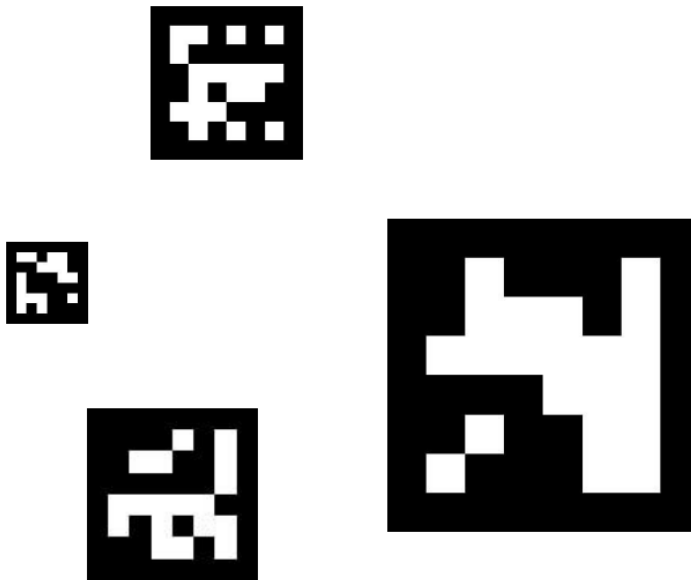
The aruco functionalities are included in:

#include <[opencv2/aruco.hpp](http://opencv2.aruco.hpp)>

Markers and Dictionaries

An ArUco marker is a synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier (id). The black border facilitates its fast detection in the image and the binary codification allows its identification and the application of error detection and correction techniques. The marker size determines the size of the internal matrix. For instance a marker size of 4x4 is composed by 16 bits.

Some examples of ArUco markers:



Example of markers images

It must be noted that a marker can be found rotated in the environment, however, the detection process needs to be able to determine its original rotation, so that each corner is identified unequivocally. This is also done based on the binary codification.

A dictionary of markers is the set of markers that are considered in a specific application. It is simply the list of binary codifications of each of its markers.

The main properties of a dictionary are the dictionary size and the marker size.

- The dictionary size is the number of markers that compose the dictionary.
- The marker size is the size of those markers (the number of bits).

The aruco module includes some predefined dictionaries covering a range of different dictionary sizes and marker sizes.

One may think that the marker id is the number obtained from converting the binary codification to a decimal base number. However, this is not possible since for high marker sizes the number of bits is too high and managing such huge numbers is not practical. Instead, a marker id is simply the marker index within the dictionary it belongs to. For instance, the first 5 markers in a dictionary have the ids: 0, 1, 2, 3 and 4.

More information about dictionaries is provided in the "Selecting a dictionary" section.

Marker Creation

Before their detection, markers need to be printed in order to be placed in the environment. Marker images can be generated using the [`generateImageMarker\(\)`](#) function.

For example, let's analyze the following call:

```
cv::Mat markerImage;  
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary\(cv::aruco::DICT\_6X6\_250\);  
cv::aruco::generateImageMarker(dictionary, 23, 200, markerImage, 1);  
cv::imwrite("marker23.png", markerImage);
```

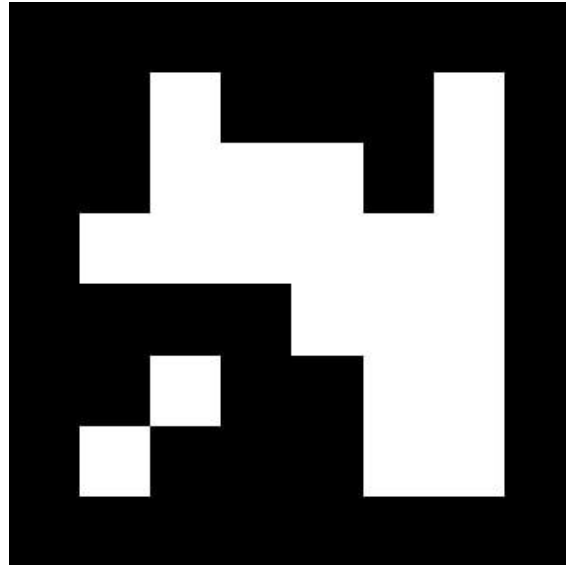
First, the Dictionary object is created by choosing one of the predefined dictionaries in the aruco module. Concretely, this dictionary is composed of 250 markers and a marker size of 6x6 bits ([`cv::aruco::DICT_6X6_250`](#)).

The parameters of `generateImageMarker` are:

- The first parameter is the Dictionary object previously created.
- The second parameter is the marker id, in this case the marker 23 of the dictionary [`cv::aruco::DICT_6X6_250`](#). Note that each dictionary is composed of a different number of markers. In this case, the valid ids go from 0 to 249. Any specific id out of the valid range will produce an exception.
- The third parameter, 200, is the size of the output marker image. In this case, the output image will have a size of 200x200 pixels. Note that this parameter should be large enough to store the number of bits for the specific dictionary. So, for instance, you cannot generate an image of 5x5 pixels for a marker size of 6x6 bits (and that is without considering the marker border). Furthermore, to avoid deformations, this parameter should be proportional to the number of bits + border size, or at least much higher than the marker size (like 200 in the example), so that deformations are insignificant.
- The fourth parameter is the output image.

- Finally, the last parameter is an optional parameter to specify the width of the marker black border. The size is specified proportional to the number of bits. For instance a value of 2 means that the border will have a width equivalent to the size of two internal bits. The default value is 1.

The generated image is:



Generated marker

A full working example is included in the `create_marker.cpp` inside the `modules/aruco/samples/`.

Note: The samples now take input from the command line using [cv::CommandLineParser](#). For this file the example parameters will look like:

```
"marker23.png" -d=10 -id=23
```

Parameters for `create_marker.cpp`:

```
const char* keys =
```

```
"{@outfile |<none> | Output image }"
```

```
"{d | | dictionary: DICT_4X4_50=0, DICT_4X4_100=1, DICT_4X4_250=2,"
```

```
"DICT_4X4_1000=3, DICT_5X5_50=4, DICT_5X5_100=5, DICT_5X5_250=6, DICT_5X5_1000=7, "
```

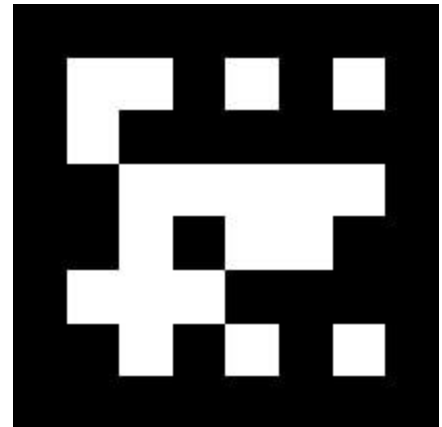
```
"DICT_6X6_50=8, DICT_6X6_100=9, DICT_6X6_250=10, DICT_6X6_1000=11, DICT_7X7_50=12,"
```

```
"DICT_7X7_100=13, DICT_7X7_250=14, DICT_7X7_1000=15, DICT_ARUCO_ORIGINAL = 16}"
```

```
"{cd | | Input file with custom dictionary }"
```

```
"{id | | Marker id in the dictionary }"  
"{ms | 200 | Marker size in pixels }"  
"{bb | 1 | Number of bits in marker borders }"  
"{si | false | show generated image }";  
}
```

Marker Detection



Given an image containing ArUco markers, the detection process has to return a list of detected markers. Each detected marker includes:

- The position of its four corners in the image (in their original order).
- The id of the marker.

The marker detection process is comprised of two main steps:

1. Detection of marker candidates. In this step the image is analyzed in order to find square shapes that are candidates to be markers. It begins with an adaptive thresholding to segment the markers, then contours are extracted from the thresholded image and those that are not convex or do not approximate to a square shape are discarded. Some extra filtering is also applied (removing contours that are too small or too big, removing contours too close to each other, etc).

2. After the candidate detection, it is necessary to determine if they are actually markers by analyzing their inner codification. This step starts by extracting the marker bits of each marker. To do so, a perspective transformation is first applied to obtain the marker in its canonical form. Then, the canonical image is thresholded using Otsu to separate white and black bits. The image is divided into different cells according to the marker size and the border size. Then the number of black or white pixels in each cell is counted to determine if it is a white or a black bit. Finally, the bits are analyzed to determine if the marker belongs to the specific dictionary. Error correction techniques are employed when necessary.

Consider the following image:

```
cv::VideoCapture inputVideo;  
  
inputVideo.open(0);  
  
cv::aruco::DetectorParameters detectorParams = cv::aruco::DetectorParameters();  
  
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT\_6X6\_250);  
  
cv::aruco::ArucoDetector detector(dictionary, detectorParams);  
  
while (inputVideo.grab()) {  
  
    cv::Mat image, imageCopy;  
  
    inputVideo.retrieve(image);  
  
    image.copyTo(imageCopy);  
  
    std::vector<int> ids;  
  
    std::vector<std::vector<cv::Point2f>> corners, rejected;  
  
    detector.detectMarkers(image, corners, ids, rejected);  
  
    // if at least one marker detected  
  
    if (ids.size() > 0)  
  
        cv::aruco::drawDetectedMarkers(imageCopy, corners, ids);  
  
    cv::imshow("out", imageCopy);  
  
    char key = (char) cv::waitKey(waitTime);  
  
    if (key == 27)
```

```
break;
```

```
}
```

Note that some of the optional parameters have been omitted, like the detection parameter object and the output vector of rejected candidates.

A full working example is included in the detect_markers.cpp inside the modules/aruco/samples/.

Note: The samples now take input from the command line using [cv::CommandLineParser](#). For this file the example parameters will look like

```
-v=/path_to_aruco_tutorials/aruco_detection/images/singlemarkersoriginal.jpg -d=10
```

Parameters for detect_markers.cpp:

```
const char* keys =
```

```
"{d | | dictionary: DICT_4X4_50=0, DICT_4X4_100=1, DICT_4X4_250=2,"
```

```
"DICT_4X4_1000=3, DICT_5X5_50=4, DICT_5X5_100=5, DICT_5X5_250=6, DICT_5X5_1000=7, "
```

```
"DICT_6X6_50=8, DICT_6X6_100=9, DICT_6X6_250=10, DICT_6X6_1000=11, DICT_7X7_50=12,"
```

```
"DICT_7X7_100=13, DICT_7X7_250=14, DICT_7X7_1000=15, DICT_ARUCO_ORIGINAL = 16,"
```

```
"DICT_APRILTAG_16h5=17, DICT_APRILTAG_25h9=18, DICT_APRILTAG_36h10=19,  
DICT_APRILTAG_36h11=20}";
```

```
"{cd | | Input file with custom dictionary }"
```

```
"{v | | Input from video or image file, if omitted, input comes from camera }"
```

```
"{ci | 0 | Camera id if input doesnt come from video (-v) }"
```

```
"{c | | Camera intrinsic parameters. Needed for camera pose }"
```

```
"{l | 0.1 | Marker side length (in meters). Needed for correct scale in camera pose }"
```

```
"{dp | | File of marker detector parameters }"
```

```
"{r | | show rejected candidates too }"
```

```
"{refine | | Corner refinement: CORNER_REFINE_NONE=0, CORNER_REFINE_SUBPIX=1,"
```

```
"CORNER_REFINE_CONTOUR=2, CORNER_REFINE_APRILTAG=3}";
```

```
}
```

Pose Estimation

The next thing you'll probably want to do after detecting the markers is to use them to get the camera pose.

To perform camera pose estimation, you need to know your camera's calibration parameters. These are the camera matrix and distortion coefficients. If you do not know how to calibrate your camera, you can take a look at the [calibrateCamera\(\)](#) function and the Calibration tutorial of OpenCV. You can also calibrate your camera using the aruco module as explained in the **Calibration with ArUco and ChArUco** tutorial. Note that this only needs to be done once unless the camera optics are modified (for instance changing its focus).

As a result of the calibration, you get a camera matrix: a matrix of 3x3 elements with the focal distances and the camera center coordinates (a.k.a intrinsic parameters), and the distortion coefficients: a vector of 5 or more elements that models the distortion produced by your camera.

When you estimate the pose with ArUco markers, you can estimate the pose of each marker individually. If you want to estimate one pose from a set of markers, use ArUco Boards (see the **Detection of ArUco Boards** tutorial). Using ArUco boards instead of single markers allows some markers to be occluded.

The camera pose relative to the marker is a 3d transformation from the marker coordinate system to the camera coordinate system. It is specified by rotation and translation vectors (see [cv::solvePnP\(\)](#) function for more information).

[cv::Mat](#) cameraMatrix, distCoeffs;

// You can read camera parameters from tutorial_camera_params.yml

readCameraParameters(cameraParamsFilename, cameraMatrix, distCoeffs); // This function is implemented in aruco_samples_utility.hpp

std::vector<cv::Vec3d> rvecs, tvecs;

// Set coordinate system

[cv::Mat](#) objPoints(4, 1, [CV_32FC3](#));

...

// Calculate pose for each marker

for (int i = 0; i < nMarkers; i++) {

[solvePnP](#)(objPoints, corners.at(i), cameraMatrix, distCoeffs, rvecs.at(i), tvecs.at(i));

}

- The markerCorners parameter is the vector of marker corners returned by the [detectMarkers\(\)](#) function.
- The second parameter is the size of the marker side in meters or in any other unit. Note that the translation vectors of the estimated poses will be in the same unit
- cameraMatrix and distCoeffs are the camera calibration parameters that were created during the camera calibration process.
- The output parameters rvecs and tvecs are the rotation and translation vectors respectively, for each of the markers in markerCorners.

The marker coordinate system that is assumed by this function is placed in the center (by default) or in the top left corner of the marker with the Z axis pointing out, as in the following image. Axis-color correspondences are X: red, Y: green, Z: blue. Note the axis directions of the rotated markers in this image.

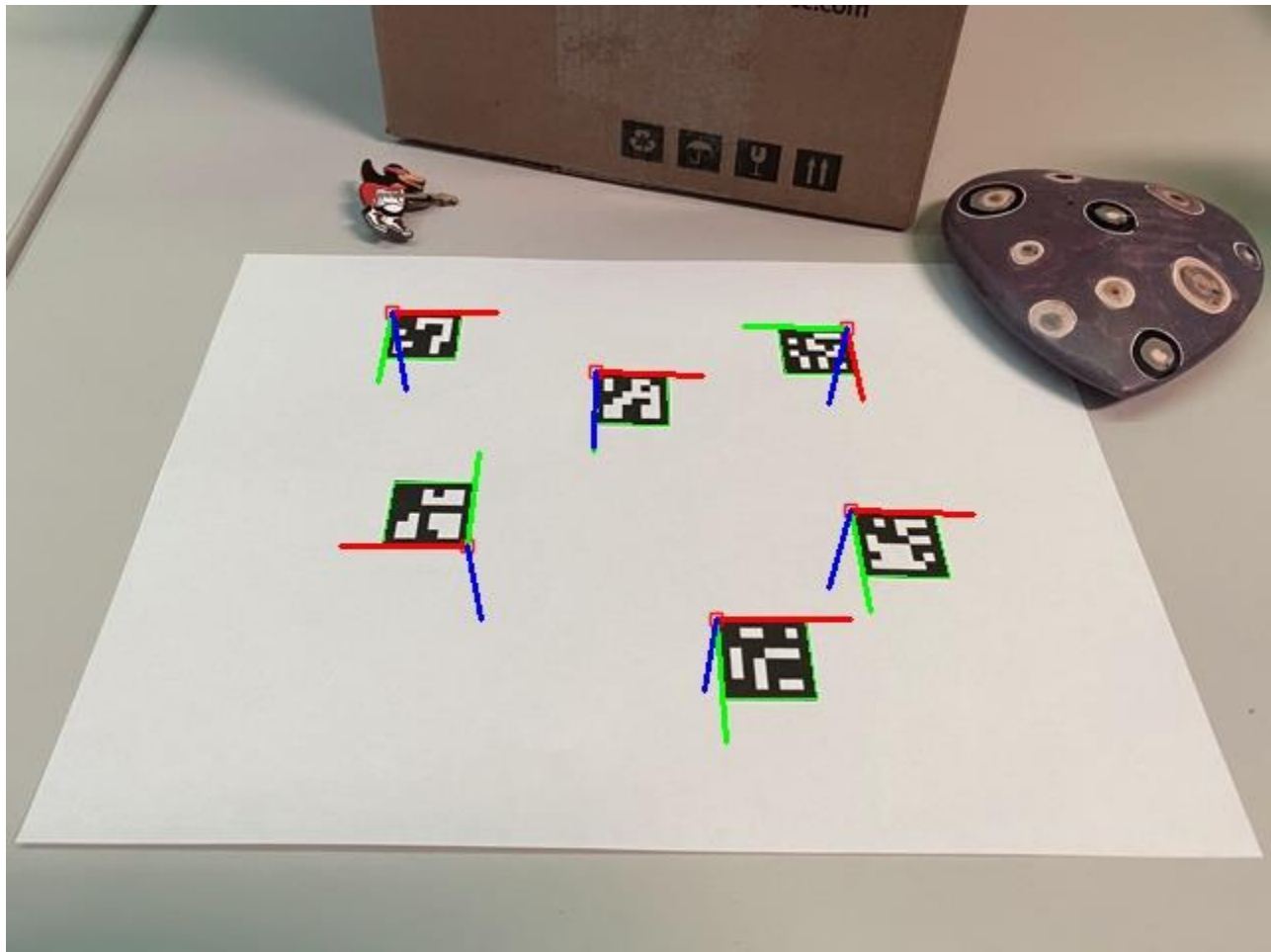


Image with axes drawn

The aruco module provides a function to draw the axis as in the image above, so pose estimation can be checked:

```
inputImage.copyTo(outputImage);

for (int i = 0; i < rvecs.size(); ++i) {

    auto rvec = rvecs[i];

    auto tvec = tvecs[i];

    cv::drawFrameAxes(outputImage, cameraMatrix, distCoeffs, rvec, tvec, 0.1);

}
```

- outputImage is the input/output image where the markers will be drawn (it will normally be the same image where the markers were detected).
- cameraMatrix and distCoeffs are the camera calibration parameters.
- rvec and tvec are the pose parameters for the marker whose axis is to be drawn.
- The last parameter is the length of the axis, in the same unit as tvec (usually meters).

A basic full example for pose estimation from single markers:

```
cv::VideoCapture inputVideo;

inputVideo.open(0);

cv::Mat cameraMatrix, distCoeffs;

float markerLength = 0.05;

// You can read camera parameters from tutorial_camera_params.yml

readCameraParameters(cameraParamsFilename, cameraMatrix, distCoeffs); // This function is
implemented in aruco_samples_utility.hpp

// Set coordinate system

cv::Mat objPoints(4, 1, CV_32FC3);

objPoints.ptr<cv::Vec3f>(0)[0] = cv::Vec3f(-markerLength/2.f, markerLength/2.f, 0);

objPoints.ptr<cv::Vec3f>(0)[1] = cv::Vec3f(markerLength/2.f, markerLength/2.f, 0);

objPoints.ptr<cv::Vec3f>(0)[2] = cv::Vec3f(markerLength/2.f, -markerLength/2.f, 0);

objPoints.ptr<cv::Vec3f>(0)[3] = cv::Vec3f(-markerLength/2.f, -markerLength/2.f, 0);
```

```
cv::aruco::DetectorParameters detectorParams = cv::aruco::DetectorParameters();  
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT\_6X6\_250);  
  
aruco::ArucoDetector detector(dictionary, detectorParams);  
  
while (inputVideo.grab()) {  
  
    cv::Mat image, imageCopy;  
  
    inputVideo.retrieve(image);  
  
    image.copyTo(imageCopy);  
  
    std::vector<int> ids;  
  
    std::vector<std::vector<cv::Point2f>> corners;  
  
    detector.detectMarkers(image, corners, ids);  
  
    // If at least one marker detected  
  
    if (ids.size() > 0) {  
  
        cv::aruco::drawDetectedMarkers(imageCopy, corners, ids);  
  
        int nMarkers = corners.size();  
  
        std::vector<cv::Vec3d> rvecs(nMarkers), tvecs(nMarkers);  
  
        // Calculate pose for each marker  
  
        for (int i = 0; i < nMarkers; i++) {  
  
            solvePnP(objPoints, corners.at(i), cameraMatrix, distCoeffs, rvecs.at(i), tvecs.at(i));  
  
        }  
  
        // Draw axis for each marker  
  
        for(unsigned int i = 0; i < ids.size(); i++) {  
  
            cv::drawFrameAxes(imageCopy, cameraMatrix, distCoeffs, rvecs[i], tvecs[i], 0.1);  
  
        }  
  
    }  
  
    // Show resulting image and close window
```

```

cv::imshow("out", imageCopy);

char key = (char) cv::waitKey(waitTime);

if (key == 27)

break;

}

```

Sample video:

A full working example is included in the detect_markers.cpp inside the modules/aruco/samples/.

Note: The samples now take input from the command line using [cv::CommandLineParser](#). For this file the example parameters will look like

```
-v=/path_to_aruco_tutorials/aruco_detection/images/singlemarkersoriginal.jpg -d=10
```

```
-c=/path_to_aruco_samples/tutorial_camera_params.yml
```

Parameters for detect_markers.cpp:

```

const char* keys =

"d | | dictionary: DICT_4X4_50=0, DICT_4X4_100=1, DICT_4X4_250=2,"

"DICT_4X4_1000=3, DICT_5X5_50=4, DICT_5X5_100=5, DICT_5X5_250=6, DICT_5X5_1000=7, "

"DICT_6X6_50=8, DICT_6X6_100=9, DICT_6X6_250=10, DICT_6X6_1000=11, DICT_7X7_50=12,"

"DICT_7X7_100=13, DICT_7X7_250=14, DICT_7X7_1000=15, DICT_ARUCO_ORIGINAL = 16,"

"DICT_APRILTAG_16h5=17, DICT_APRILTAG_25h9=18, DICT_APRILTAG_36h10=19,
DICT_APRILTAG_36h11=20}"

"{cd | | Input file with custom dictionary}"

"{v | | Input from video or image file, if omitted, input comes from camera}"

"{ci | 0 | Camera id if input doesnt come from video (-v) }"

"{c | | Camera intrinsic parameters. Needed for camera pose}"

"{l | 0.1 | Marker side length (in meters). Needed for correct scale in camera pose}"

"{dp | | File of marker detector parameters}"

"{r | | show rejected candidates too}"

```

```
"{refine | | Corner refinement: CORNER_REFINE_NONE=0, CORNER_REFINE_SUBPIX=1,"  
"CORNER_REFINE_CONTOUR=2, CORNER_REFINE_APRILTAG=3}";  
}
```

Note

To work with examples from the tutorial, you can use camera parameters from `tutorial_camera_params.yml`. An example of use in `detect_markers.cpp`.

Selecting a dictionary

The `aruco` module provides the `Dictionary` class to represent a dictionary of markers.

In addition to the marker size and the number of markers in the dictionary, there is another important parameter of the dictionary - the inter-marker distance. The inter-marker distance is the minimum distance between dictionary markers that determines the dictionary's ability to detect and correct errors.

In general, smaller dictionary sizes and larger marker sizes increase the inter-marker distance and vice versa. However, the detection of markers with larger sizes is more difficult due to the higher number of bits that need to be extracted from the image.

For instance, if you need only 10 markers in your application, it is better to use a dictionary composed only of those 10 markers than using a dictionary composed of 1000 markers. The reason is that the dictionary composed of 10 markers will have a higher inter-marker distance and, thus, it will be more robust to errors.

As a consequence, the `aruco` module includes several ways to select your dictionary of markers, so that you can increase your system robustness:

Predefined dictionaries

This is the easiest way to select a dictionary. The `aruco` module includes a set of predefined dictionaries in a variety of marker sizes and number of markers. For instance:

```
cv::aruco::Dictionary dictionary = cv::aruco::getPredefinedDictionary(cv::aruco::DICT\_6X6\_250);
```

[cv::aruco::DICT_6X6_250](#) is an example of predefined dictionary of markers with 6x6 bits and a total of 250 markers.

From all the provided dictionaries, it is recommended to choose the smallest one that fits your application. For instance, if you need 200 markers of 6x6 bits, it is better to use [cv::aruco::DICT_6X6_250](#) than [cv::aruco::DICT_6X6_1000](#). The smaller the dictionary, the higher the inter-marker distance.

The list of available predefined dictionaries can be found in the documentation for the `PredefinedDictionaryType` enum.

Automatic dictionary generation

A dictionary can be generated automatically to adjust the desired number of markers and bits to optimize the inter-marker distance:

```
cv::aruco::Dictionary dictionary = cv::aruco::extendDictionary(36, 5);
```

This will generate a customized dictionary composed of 36 markers of 5x5 bits. The process can take several seconds, depending on the parameters (it is slower for larger dictionaries and higher numbers of bits).

Manual dictionary generation

Finally, the dictionary can be configured manually, so that any encoding can be used. To do that, the Dictionary object parameters need to be assigned manually. It must be noted that, unless you have a special reason to do this manually, it is preferable to use one of the previous alternatives.

The Dictionary parameters are:

```
class Dictionary {  
  
public:  
  
    cv::Mat bytesList; // marker code information  
  
    int markerSize; // number of bits per dimension  
  
    int maxCorrectionBits; // maximum number of bits that can be corrected  
  
    ...  
}
```

`bytesList` is the array that contains all the information about the marker codes. `markerSize` is the size of each marker dimension (for instance, 5 for markers with 5x5 bits). Finally, `maxCorrectionBits` is the maximum number of erroneous bits that can be corrected during the marker detection. If this value is too high, it can lead to a high number of false positives.

Each row in `bytesList` represents one of the dictionary markers. However, the markers are not stored in their binary form, instead they are stored in a special format to simplify their detection.

Fortunately, a marker can be easily transformed to this form using the static method `Dictionary::getBytesListFromBits()`.

For example:

```

cv::aruco::Dictionary dictionary;

// Markers of 6x6 bits

dictionary.markerSize = 6;

// Maximum number of bit corrections

dictionary.maxCorrectionBits = 3;

// Let's create a dictionary of 100 markers

for(int i = 0; i < 100; i++)

{

// Assume generateMarkerBits() generates a new marker in binary format, so that

// markerBits is a 6x6 matrix of CV_8UC1 type, only containing 0s and 1s

cv::Mat markerBits = generateMarkerBits();

cv::Mat markerCompressed = cv::aruco::Dictionary::getByteListFromBits(markerBits);

// Add the marker as a new row

dictionary.bytesList.push_back(markerCompressed);

}

```

Detector Parameters

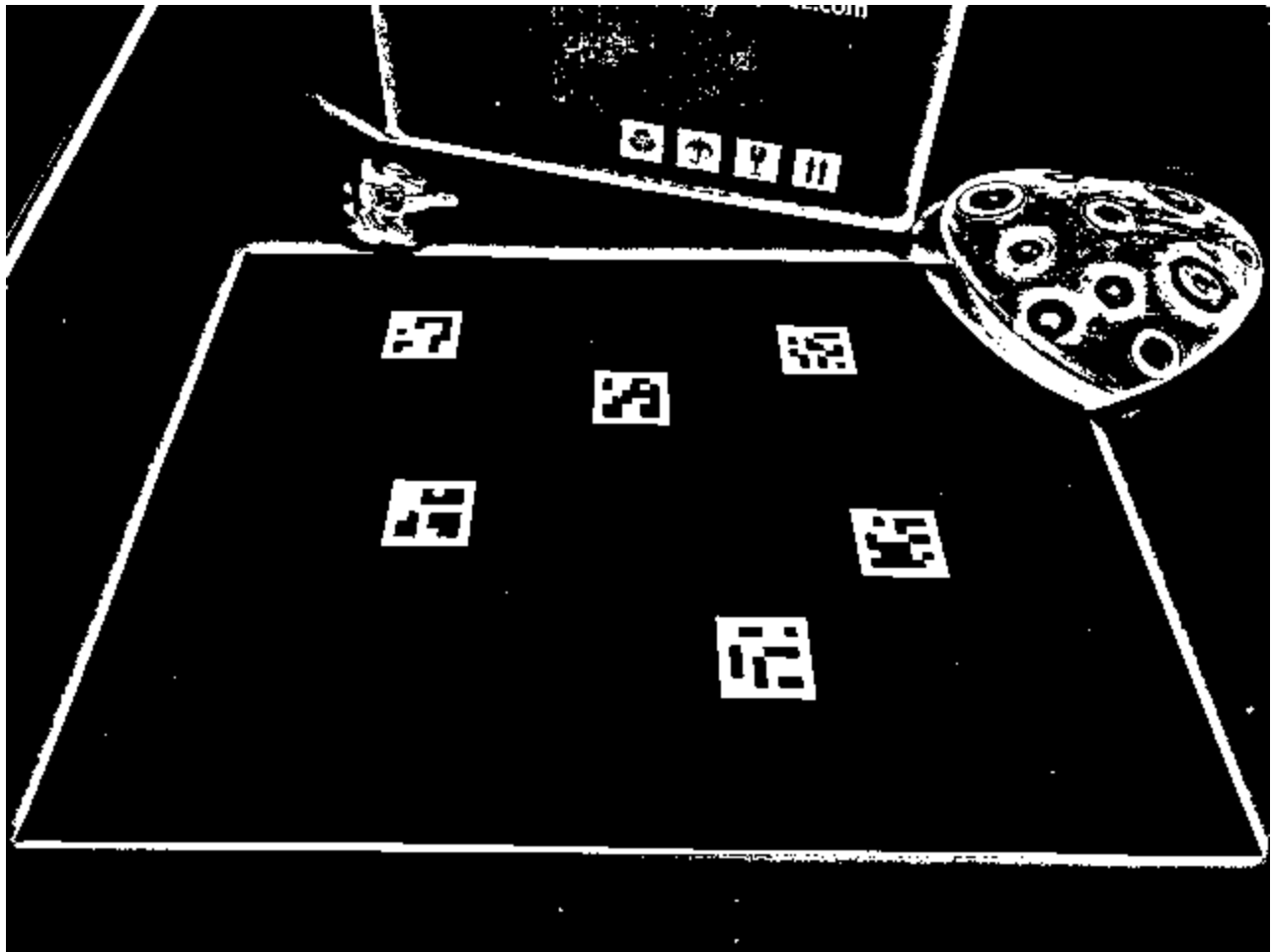
One of the parameters of ArucoDetector is a DetectorParameters object. This object includes all the options that can be customized during the marker detection process.

This section describes each detector parameter. The parameters can be classified depending on the process in which they're involved:

Thresholding

One of the first steps in the marker detection process is adaptive thresholding of the input image.

For instance, the thresholded image for the sample image used above is:



Thresholded image

This thresholding can be customized with the following parameters:

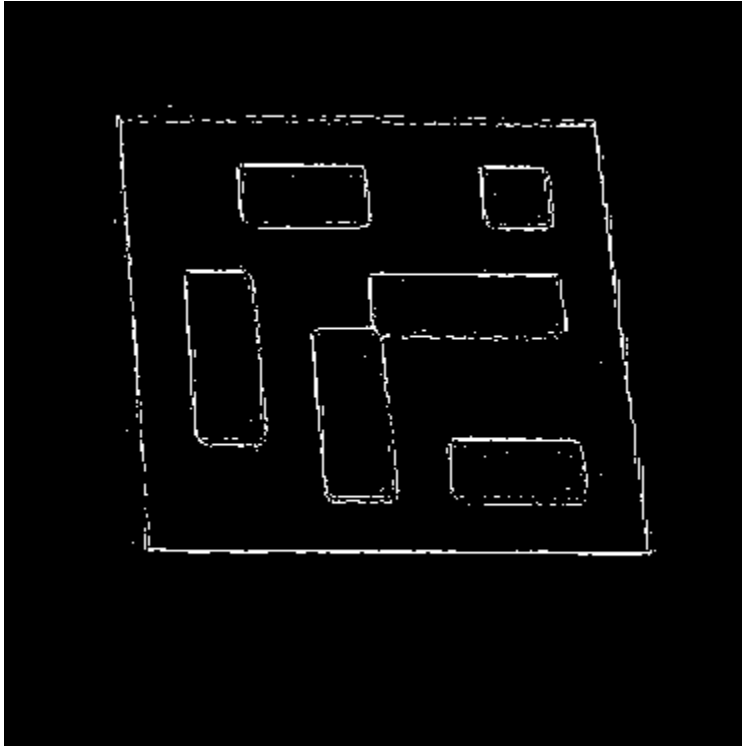
adaptiveThreshWinSizeMin, adaptiveThreshWinSizeMax, and adaptiveThreshWinSizeStep

The `adaptiveThreshWinSizeMin` and `adaptiveThreshWinSizeMax` parameters represent the interval where the thresholding window sizes (in pixels) are selected for the adaptive thresholding (see OpenCV [threshold\(\)](#) function for more details).

The parameter `adaptiveThreshWinSizeStep` indicates the increments of the window size from `adaptiveThreshWinSizeMin` to `adaptiveThreshWinSizeMax`.

For instance, for the values `adaptiveThreshWinSizeMin = 5` and `adaptiveThreshWinSizeMax = 21` and `adaptiveThreshWinSizeStep = 4`, there will be 5 thresholding steps with window sizes 5, 9, 13, 17 and 21. On each thresholding image, marker candidates will be extracted.

Low values of window size can "break" the marker border if the marker size is too large, causing it to not be detected, as in the following image:



Broken marker image

On the other hand, too large values can produce the same effect if the markers are too small, and can also reduce the performance. Moreover the process will tend to global thresholding, resulting in a loss of adaptive benefits.

The simplest case is using the same value for `adaptiveThreshWinSizeMin` and `adaptiveThreshWinSizeMax`, which produces a single thresholding step. However, it is usually better to use a range of values for the window size, although many thresholding steps can also reduce the performance considerably.

Default values: